Modular and Hierarchical Discrete Control for Applications and Middleware Deployment in IoT and Smart Buildings

Adja Ndeye Sylla*,Maxime Louvel†, Eric Rutten‡ and Gwenaël Delaval‡
* Univ. Grenoble Alpes, CEA, LETI, DACLE, LIALP, F-38000 Grenoble
AdjaNdeye.Sylla@cea.fr
† Bag-Era
Maxime.Louvel@bag-era.fr
‡ Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, F-38000 Grenoble
Eric.Rutten@inria.fr, Gwenael.Delaval@inria.fr

***Abstract*— In the Internet of Things (IoT) and Smart Homes and Buildings, sensors and actuators are controlled through a management software, that runs on a distributed network of heterogeneous processors. Such management systems have to be self-adaptive w.r.t. different aspects, at applications level (functionalities) as well as deployment level (software tasks, execution platform). Holding a well-mastered and safe behaviour of the overall system, in presence of these concurrent adaptations, is a complex control problem. We approach this problem by applying techniques from the area of Supervisory Control for Discrete Event Systems (DES), where the space of configurations at the different levels are modeled with automata. We use programming language support tools, Heptagon/BZR and ReaX, to build up a design environment for the considered application domain. This paper contributes with (i) generic behavioural models for both the applicative and deployment aspects of systems; (ii) applications of Discrete Controller Synthesis (DCS) to design controllers, especially modular and hierarchical control structures; (iii) an implemented case study.**

Keywords : Discrete Event Systems, Application, Software Systems, Smart Building

I. Introduction

A. Internet of Things and Smart Buildings

Today's computer-based systems (e.g., smart buildings and homes) must adapt to their changing environment in order to remain operational and achieve their objectives. For this purpose, they continuously collect data, make decisions and execute reconfiguration commands. Depending on the collected data, the commands executed by a system are related to its applicative functional logic and also to the management of the computing system itself, typically the deployment of software objects on an execution platform.

Such management systems have to be self-adaptive w.r.t. different concurrent aspects. Applications have to adapt their functionalities to the environment conditions (e.g., temperature, presence). Software tasks implementing them have to be deployed on the execution platform, adapting on available resources (e.g., starting/stopping or migrating objects). The execution platform itself has to adapt to variations in workload or availability (e.g., switching on/off nodes, fault tolerance). Therefore, managing these different concurrent adaptations in order to hold a well-mastered and safe behaviour of the overall system is a complex control problem.

The design and the implementation of such a system raises problems of avoiding decisions that can be conflicting, useless or violate other target objectives. In the literature, solutions have been proposed for the design and the implementation of reliable adaptive systems [5], [7], [17], [11], [12]. However, they focus either on the systems application logic or on the software deployment but not on both together. The work in [11], [12] use Petri nets and perform model checking, we propose a supervisory control based approach.

B. Application of DES control with Heptagon/BZR

To overcome these problems, we propose a generic control support for reliable adaptive systems. We apply techniques from the area of Supervisory Control for Discrete Event Systems (DES), where the space of configurations at the different levels (application, deployment, execution platform) are modelled with transitions systems, particularly automata.

We use programming language support tools, Heptagon/BZR [4] and ReaX [2], and their Discrete Controller Synthesis (DCS) capabilities, to generate correct executable controllers. These can be integrated in the underlying middleware, based on the transactional rule-based framework LINC [9], to build up a design environment for our application domain. This allows to (i) ensure the behavioural reliability of systems (absence of conflicting or incorrect decisions), (ii) handle their heterogeneity and (iii) guarantee their execution reliability (absence of inconsistencies). An inconsistency is due to a hardware failure or a communication error and it occurs when the system assumes that a command is executed but it actually is not for instance due to a hardware failure.

We are following the same approach of DES model-based design of reconfiguration controllers on different classes of computing systems. We consider small, embedded hardware systems, such as FPGA [10] as well as larger, more distributed software systems like web servers in the Cloud [3], [1], or heterogeneous platforms, composed of numerous devices with very different computation power, operating systems and communication protocols, like in the domain of the Internet of Things (e.g., smart buildings [16]). These works are distinguished by their very different target computing systems, w.r.t. execution platforms and application domains: therefore they bring complementary experience, towards the

identification of generic modelling methods, in order to pose and solve logical control problems in computing systems.

C. Contributions of the paper

This paper contributes in the following points:

1) behavioural automata-based modelling, and objectives formalisation, for the coordination of the applicative and deployment mangers of smart building systems;
2) applications of Discrete Controller Synthesis to design monolithic or modular and hierarchical controllers;
3) an implemented case study for a smart office, including controller execution in a reactive autonomic loop, a transactional runtime and an abstraction layer.

The paper is structured as follows. Section II gives the background notions. Then, Section III presents the target domain of smart buildings. Section IV describes behavioural models and objectives. Section V presents a case study. Finally, Section VI discusses and gives the perspectives.

II. Background

A. Internet of Things and Smart buildings

The application domain we target is smart buildings, in the context of the Internet of Things (IoT), equipped with heterogeneous devices and managed by a software system.

1) Considered aspects: a smart building system consists of two managers that interact to achieve a set of objectives. An *applicative manager* handles the sensors and the actuators of the building and provides applicative functionalities e.g., presence detection, access control and luminosity control. This manager is implemented as a set software entities that achieve applicative objectives. A *deployment manager* handles these entities, and their deployment on the execution platform of the building. It consists of a middleware that switches on/off computing devices, starts/stops/migrates software entities and installs software resources if required. The aim is to achieve a set of deployment objectives (e.g., minimize the energy consumption of the execution platform).

These two managers can interact in four ways. The application can affect the deployment: for instance, providing a new applicative functionality, with a specified Quality of Service (QoS) level, can require switching on additional computing devices. The deployment can affect the application: for instance, the overload of a computing device can require decreasing the QoS provided by the application. The application can change without affecting the deployment: for instance, the application can request for a functionality that is provided by a set of software entities already deployed (for another functionality). The deployment can change without affecting the application: for instance, the deployment can switches off unused computing devices for energy savings.

This paper models each manager to handle their coordination and solve the logical control problems they raise.

2) Logical control problems encountered: for both aspects, to achieve its target objectives, a smart building system computes and executes at each step a set of commands that could be conflicting or violate other target objectives. A conflict occurs when a device or a software entity receives at the same instant contradictory commands. An objective is violated when it is not met due the execution of one or several commands. For the applicative aspect, both conflicts and objectives violations can be caused by environment dependencies and are in this case, difficult to detect. For instance, opening the window of a room for natural ventilation can introduce noise and violate an objective that consists in maintaining the noise level below a given threshold. Conflicts and objectives violations could lead to unsafe or undesired states that should be avoided. The smart building system must also autonomously adapt to its environment by reacting to changes. We follow the approach of autonomic computing [6] for performing self-configuration, self-optimization, self-healing or self-protection, through a control loop. In this paper, to deal with the logical control problems, we use Discrete Controller Synthesis (DCS) [13] through a reactive language and tool, and allow for executable code generation.

B. Reactive language for Discrete Controller Synthesis

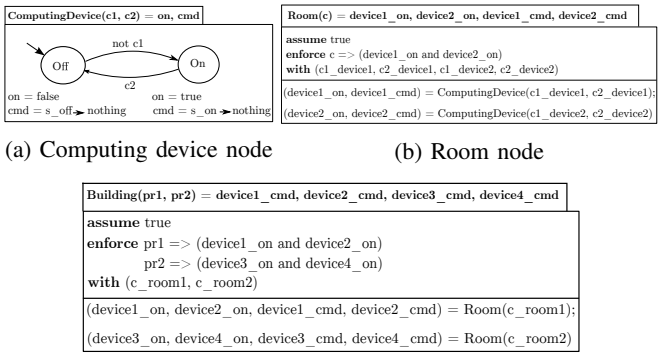
The reactive language used is Heptagon/BZR (H/BZR) [4]. It is based on a contract mechanism and allows for declarative design of controllers trough discrete controller synthesis.

1) Design and synthesis of a controller: A H/BZR program is first written, for the considered system, as a set of hierarchical blocks called nodes. Each node declares input and output *flows*, i.e., sequences of values taken by inputs and outputs on a discrete scale of time. A node can contain *equations* and control structures on these equations, like automata. Each automaton models an entity of the system and they can be composed in parallel or in hierarchy. A node can be provided with a *contract* that define objectives to be enforced in the program through DCS. Once designed, the program is compiled to generate the system controller.

a) Equations: they define outputs, and possible local variables, in terms of other variables (inputs, outputs, local variables). Equations can be composed of memorised values, introduced by the **`pre`** delay operator: **`pre`** `e` holds the value of `e` at the previous instant. Memories can be initialized with the `->` operator: `c` **`->pre`** `e` holds the value `c` at the first instant, and the previous value of `e` at following ones.

b) Automata: they are control structures contained in nodes and allowing to associate *equations* (or hierarchically, sub-automata) to *states* of the system. An automaton has a set of states, one of them being the initial state, and transitions between them. States are associated to equations that give values to the output flows of the node that contains the automaton. At each instant, one state is activated and a value must be defined for each output flow. A transition goes from one state to another and is associated to a boolean expression that is related to one or more flows of the automaton node.

Fig. 1a, presents an automaton designed for the control of a computing device. This automaton is contained in a node that has two input flows (`c1`, `c2`) and two output flows (`on`, `cmd`). The automaton has two states (`Off`, `On`) and two transitions. Each state is associated to two equations that give values to the output flows of the node. In the state `Off`, the output flow `on` is equal to `false`. This meaning that the



(a) Computing device node (b) Room node

(c) Building node

Fig. 1: Example of H/BZR program

computing device is off. In the state `Off`, the output flow `cmd` is equal to `s_off` (switch off) when this state is newly reached and `nothing` otherwise. The reason is twofold. First, the value of an output flow must be defined at each instant. Second, this prevents from, continuously, sending `cmd = s_off` while the computing device is already switched off. The initial state of the automaton is `Off`. When the input flow `c1` is false, the automaton goes to the state `On` and the output flows takes the values given by the equations of this state. Otherwise (`c1` is true), the automaton remains in `Off`.

c) Contract*:* it allows for DCS through a tool called the ReaX [2], and consists of three parts: `assume`, `enforce` and `with`. The `assume` defines the hypotheses of the considered system. The `enforce` defines the target objectives (invariance properties). The `with` declares the controllable variables, that will be used to enforce the objectives. From a *contract*, the DCS algorithm computes the possible values of the controllable variables. This is done by exploring the state space of the model and inhibiting all the behaviours that violate the target objectives. The aim is to enforce all the objectives whatever the uncontrollable variables values. After the synthesis, several solutions can be possible regarding the target objectives. However, one solution must be selected. This selection is done by the backend of the H/BZR compiler by favouring the value true to false for each boolean controllable variable following the order of their declaration.

Fig. 1b presents an example of node with *contract* to switch on both computing devices of a room when a presence is detected. This node defines two instances of the computing device node (Fig. 1a) and composes, in parallel, their automata using the operator `;`. The *contract* defines no hypothesis (`assume true`), one objective and four controllable variables that are the input of the computing device nodes. The objective specifies that when `c` is true (a presence is detected in the room), both computing devices must be on.

d) Cost of the controller synthesis: the DCS algorithm has a cost exponential in the number of variables used in the model of the system [4]. These variables consists of the state variables, the controllable and the uncontrollable variables. Hence, for a system that consists of a high number of entities, the controller synthesis can take a lot of time or not succeed due to CPU and/or RAM limitations. For such systems, H/BZR allows for performing the controller synthesis modularly to reduce the related synthesis cost.

2) Modular design and synthesis of a controller: this consists in dividing the system into subsystems and defining for each subsystem, a node with a *contract*. This node instantiates and composes relevant automata and enforce a subset of the target objectives. In this case, DCS is performed on each subsystem instead on the whole system. This decreases the DCS execution time and its resources consumption. This also allows for the generation of several controllers, one for each subsystem, instead of a single one for the whole system.

Fig. 1c shows an example of modular design and synthesis of a controller. The `Room` node (Fig. 1b) is instantiated twice, to model a building of two rooms. This `Building` node takes as input two flows `pr1` and `pr2`, that model the value of a presence sensor in each room. Then, this node is provided with a global *contract*, to achieve the following objective: for each room, both computing devices must be switched on when a presence is detected by the associated sensor. The *contract* defines two objectives and two controllable variables. This enables the modular synthesis of a controller.

3) Execution of a controller: the compilation of a H/BZR program (e.g., Fig. 1) generates a `step` function. In case of modular DCS, several `step` are generated (one for each node with a *contract*) but one of them is the main `step`. The `step` takes as parameter the inputs, computes the outputs and updates the state of the automaton modelling the system. One execution of the `step` corresponds to one reaction of the system. Hence, the `step` must be executed each time a relevant event occurs. Executing the `step` requires consistency between the automaton state and the actual system automaton. For this, a transactional middleware (LINC [8]) is used and the `step` is invoked in a LINC rule as detailed in [14], for its execution in the form of a reliable autonomic loop [16]. Fig. 2 illustrates this loop, with an abstraction layer that hides the heterogeneity of the system, the controller to execute and a transactional execution mechanism that avoids inconsistencies. Data are first collected through the abstraction layer. Then, they are analysed by the controller which computes correct and coherent commands that are executed by the transactional mechanism through the abstraction layer.

III. Description of a smart building system

This section identifies informally, for both the applicative and deployment aspects of a smart building system, the controlled entities, the relevant events and the target objectives.

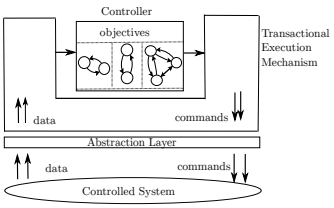


Fig. 2: Architecture of a reliable loop

A. Applicative aspect

1) Controlled entities: they consists of the actuators that are installed in the considered building. Examples of such actuators are lamps, shutters, windows, doors and heaters.

2) Relevant events: they consist of data that are produced by the sensors of the building. These data are related to the states of the actuators (e.g., opening of a window or a door) or the building (e.g., presence) and to the environment parameters (e.g., temperature, noise, CO_2). The events also consists of information about meetings (e.g., feature), about time (e.g., 7 AM) and requests for maintenance of actuators.

3) Target objectives: an applicative objective specifies, depending on the events that occur in the system, the value that must be taken by an environment parameter or by the state of an actuator. Applicative objectives are of three types

- **a_obj$_{\mathbf{T1}}$**: keep the value of an environment parameter in a specified interval (delimited by th_{min} and th_{max});
- **a_obj$_{\mathbf{T2}}$**: avoid the situation where an actuator that is requested to be idle for a maintenance operation, can be not used but remains used by the application;
- **a_obj$_{\mathbf{T3}}$**: decrease (resp. increase) the value of an environment parameter when its actual value (measured by a sensor) is below (resp. above) a specified threshold.

B. Deployment aspect

1) Controlled entities: they consist of the software entities, implemented to offer the target applicative functionalities, and the execution platform of the considered building.

The software entities consist of tasks, objects and rules. A task offers one or more applicative functionalities, using Input/Output (I/O) resources. Each task has a set of versions, with different QoS (`High`, `Medium`, `Low`), and transitions specifying the changes of versions that are possible. For instance, let us consider a task with three versions V_1, V_2 and V_3. For this task, it can be valid to go from V_1 to V_2 and from V_2 to V_3 but not from V_1 to V_3, for instance for sequencing reasons. A task version consists of rules and configurations of objects. A rule communicates with one or several objects and is executed by an object. An object can have required I/O resources and a set of configurations. Both a rule and an object configuration have a CPU (resp. a RAM) load, that is either negligible or expressed as a percentage (resp. in MB).

The execution platform is composed of a set of computing devices and I/O resources. A computing device consists of a host and a set of I/O resources. Hosts are heterogeneous, they have different capacities in terms of CPU and RAM. An I/O resource is either software or hardware. Both have a type (e.g., USB dongles, OpenCV) and a hardware I/O resource has a usage mode that specifies an usage constraint. The usage mode of a resource is equal to `R` (reading), `1W` (writing by one entity) and `nW` (writing by several entities).

2) Relevant events: they consist of data related to load variations, to the execution platform and those measured by the sensors of the building. Examples of events are failures of hardware I/O resources, maintenance requests of computing devices or their loads. The events can also be related to time.

3) Target objectives: they are related to the considered building execution platform and to the deployment of the software entities. Deployment objectives are of five types

- **d_obj$_{\mathbf{T1}}$**: allocate to objects, the computing devices and the hardware and/or software I/O resources they require. This means that when an object is started, its computing device must be on and must have its required resources;
- **d_obj$_{\mathbf{T2}}$**: ensure the constraints related to the usage modes (exclusive ot not) of the hardware I/O resources;
- **d_obj$_{\mathbf{T3}}$**: avoid the situation where a computing device that is requested for maintenance is not switched off;
- **d_obj$_{\mathbf{T4}}$**: if possible, decrease the load of a computing device when its actual load, that is measured, is above a maximum threshold. Similarly, increase the load of a computing device or switch it off when its actual load is below a minimum threshold. The aim is to avoid both overloading and under-loading the computing devices;
- **d_obj$_{\mathbf{T5}}$**: minimize the execution platform energy consumption, by using the minimum of computing devices.

C. Coordination of both aspects

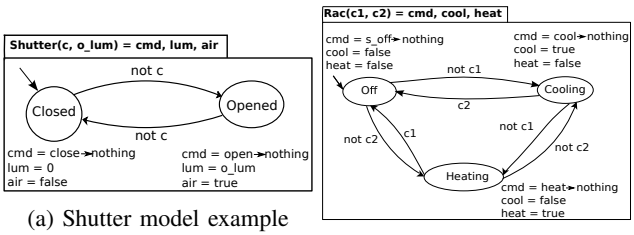
To handle their interactions, the applicative and the deployment managers must be coordinated. The coordination scheme and its objectives are presented in the followings.

1) Coordination scheme: it consists of three points

- when the applicative manager requests the starting of a functionality, with a given QoS, a task version is first selected among those that offer the functionality and have a version with the requested QoS. Then, a notification is sent to the deployment manager which deploys the objects and rules of the selected task version on the execution platform. The selection of the task version is done based on information about the tasks (offered functionalities, versions and their QoS) and on the status of the execution platform (computing devices and I/O resources). These objects and rules are kept deployed as long as the functionality has to be provided;
- when the deployment manager can no longer provide the requested QoS or functionality due to the lack of computing or I/O resources in the execution platform, it sends a notification. In this case, the application manager can, for instance, ask for a another QoS or functionality that has a cheaper resource consumption;
- when the application managers requests the stopping of a functionality, the deployment managers is first notified. Then, it stops the associated objects and rules. The deployment manager can also switch off the computing devices that are no longer used, for energy savings.

2) Coordination objectives: they consist of three types

- **c_obj$_{\mathbf{T1}}$**: when a functionality is requested to be provided, one version of a task among the set of tasks that offer this functionality must be active, if this is possible;
- **c_obj$_{\mathbf{T2}}$**: for a functionality, when no task that offers it is active, its applicative objectives must not be achieved;
- **c_obj$_{\mathbf{T3}}$**: when a task version is active, its objects must be started and its rules must be activated and executed.



(a) Shutter model example

(b) RAC model example

Fig. 3: Shutter and RAC models

IV. Modelling as a discrete control problem

This section presents how the coordination of a smart building managers is written as a discrete control problem and how it is solved to generate an executable controller.

A. Behavioural modelling

Each controlled entity is modelled as an automaton with points of controllability (variable starting with `c` by convention). A preference can be defined between the states of an automaton, through the declaration order of these variables.

1) Applicative aspect: Each actuator is modelled as an automaton contained in a H/BZR node. Such an automaton specifies the different states of the modelled actuator, its states transitions and its effects on the environment. For this, a set of environment parameters (e.g., luminosity, noise, air) are first defined. Then, the effects of each actuator on these environment parameters are identified and specified. This is done in the form of equations that are encapsulated in the states of the automaton. These equations have boolean or numerical variables and define their values. In the followings a set of automata modelling different actuators are presented.

a) Shutter modelling: Fig. 3a presents an automaton modelling a shutter. This automaton is contained in a node that has two input flows (`c`, `o_lum`) and three output flows (`cmd`, `lum`, `air`). The automaton has two states (`Closed`, `Opened`) and two transitions. Each state is associated to three equations to produce the command of the shutter (`cmd`) and specify its effects on the environment (`lum`, `air`). In `Closed`, the command is equal to `close` (resp. `nothing`) if this state is (resp. not) newly activated. This prevents from continuously sending the command `close` while the shutter is already `Closed`. In this state, the shutter provides a luminosity equal to zero (`lum = 0`) and does not allow outdoor air to pass (`air` = false). In `Opened`, the shutter provides a luminosity equal to the outdoor luminosity (`lum = o_lum`) and allows outdoor air to pass (`air` = true). The transitions going from a state to a different one are associated to `not c`, to open/close the shutter only when necessary.

b) Reversible Air-Condition (RAC) modelling: Fig. 3b presents an automaton modelling a RAC. Each state is associated to three equations to produce the command of the RAC and also specify its effects on the environment. For instance, in `Off`, the RAC does not cool nor heat the room. This automaton is contained in a node that has two input flows `c1` and `c2`. The reason is that, at each state, three transitions can be triggered (i.e., two transitions that leave the state and one that allows to stay). To associate a different boolean expression to each of the three transitions of a state, at least two variables are needed. Here, when the state `Off` is activated, if the input `c1` flow is `false`, the RAC automaton goes to the state `Cooling`. If `c2` is `false`, it goes to `Heating`. If both `c1` and `c2` are `true`, it remains in the state `Off`. Finally, if both `c1` and `c2` are `false`, at the same instant, the transition that was first declared is chosen. Associating `not c1` and `not c2` (resp. `c1` and `c2`) to the transitions leaving (resp. coming to) the state `Off` means that it is preferred to maintain the RAC `Off` for energy savings.

We similarly define models for lamps (and their luminosity), Mechanical Ventilation (MV) (affecting CO_2), windows (affecting heating, cooling, ventilation, pollution and noise), doors (affecting noise). These models are detailed in [15].

2) Deployment aspect: An automaton is designed for task, object, rule, I/O resource, host and computing device.

a) I/O resources modelling: the automaton of a hardware or a software I/O resource has three states (`Unavailable`, `Unused`, `Used`) and is contained in a node that has five inputs (`avail`, `fail`, `c1`, `c2` and `c3`). The input `avail` (resp. `fail`) allows to know if the I/O resource is available (resp. failed). The inputs `c1`, `c2` and `c3` are points of controllability that enable changing the state of the I/O resource. For a software resource, the input `fail` is a constant that is equal to `false` (a software does not fail). The input `c2` allows installing the software when it is equal to `false`. For a hardware resource, `c2` is a constant equal to `true` because a hardware resource cannot be installed.

Fig. 4 presents the automaton of a hardware I/O resource that is a dongle. The outputs of the automaton node respectively correspond to state of the dongle and its command.

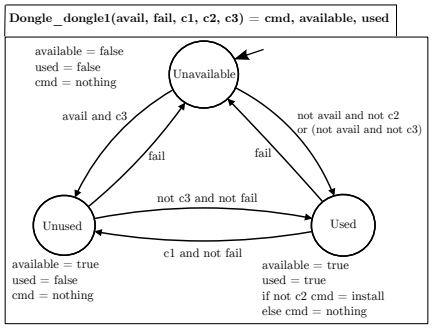


Fig. 4: Hardware resource model example

b) Host modelling: the automaton of a host has four states (`Off`, `On`, `Wait`, `Unavailable`). This automaton has six inputs (`avail`, `fail`, `rmaint`, `c1`, `c2`, `c3`). The first three inputs respectively specify if the host is available, if it is failed or if a maintenance request is sent by the maintenance team to no longer use it. The inputs `c1`, `c2`, `c3` are points of controllability. They allow for starting the host, stopping it or bringing it in the state `Wait` (upon the occurrence of a maintenance request that cannot be satisfied). In the state `Wait`, the host goes to the state `Unavailable` when it is failed or can be no longer used (allowing its maintenance).

Fig.5 presents the automaton of a host (H1). The outputs of the node that contains this automaton are the status of the host (on, available) and the command to send (cmd).

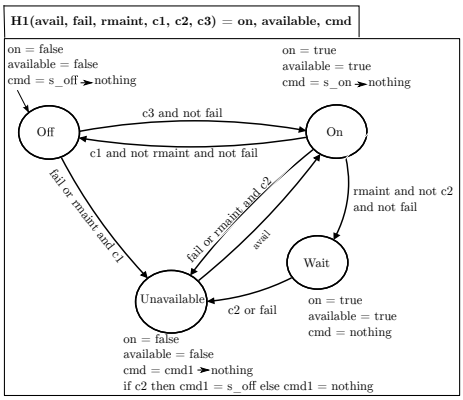


Fig. 5: Host model example

c) Computing devices modelling: the automaton of a computing device is the composition following automata:

- a host automaton for associated with the associated host;
- a hardware I/O resource automaton for each hardware I/O resource that is local to the computing device;
- software I/O resource automaton for each one of the considered software I/O resource in order to install it if it is not available on the computing device, if needed.

Fig. 6 presents an automaton modelling a computing device (D1) that has a dongle (dongle1). This automaton is the parallel composition of two automata modelling the host H1 (Fig. 5) and the hardware resource dongle1 (Fig. 4).

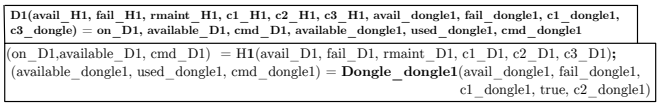


Fig. 6: Computing device model example

We similarly define models for tasks (that can be activated), objects (that can be migrated), rules (that can be deactivated). These models are presented in details in [15].

3) Coordination of both aspects: here, the fact that each applicative functionality can be requested to be provided with a given QoS or stopped is modelled. For this, an automaton is designed for each functionality, with a state for each QoS level, and an input for requesting it, as well as points of controllability for changing the current QoS or stopping.

Fig. 7 presents an example of automaton modelling a luminosity control functionality, with QoS High or Medium, points of controllability (c1, c2) and requests (req_H, req_M). When req_H is true, depending on the values of c1 and c2, the functionality is provided appropriately. The fact that c1 is declared before c2 allows for choosing the Medium QoS only if the High cannot be provided when requested.

B. Objectives formalisation

1) Applicative objectives: from Section III-A.3 are

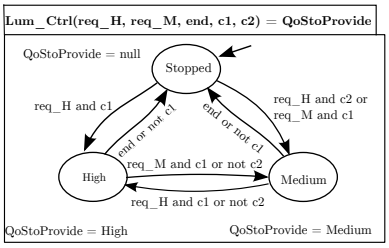


Fig. 7: Functionality model example

- **a_obj$_{T1}$**: events ⇒(var > th_{min} **and** var < th_{max}) ;
- **a_obj$_{T2}$**: for act_i in actuator set : possible ⇒ (act_i.requestMaint ⇒ act_i.unused) ;
- **a_obj$_{T3}$**: possible ⇒ ((param_measuredValue > th) ⇒ (param_compValue < **pre** param_compValue)). where param_compValue is the value computed for param in the current instant, **pre** param_compValue is the value computed in the previous instant. param_measuredValue is an input giving the actual measured value (from the environment).

Objective **a_obj$_{T3}$** is logico-numerical, as it involves a numeric input (param_measuredValue) and a numeric state (**pre** param_compValue). Achieving an objective of this type requires the use of advanced discrete control techniques, involving abstract interpretation on abstract domains such as intervals or convex polyhedra [2].

2) Deployment objectives: from Section III-B.3 are e.g.,

- **d_obj$_{T1}$**: for obj_i in objects set. obj_i.device_dest.on is the computing device selected to start or migrate obj_i: obj_i.started ⇒ (obj_i.device_dest.on **and** obj_i.requiredResources areContainedIn obj_i.device_dest.availableResources) ;
- **d_obj$_{T4}$**: for every device dev_i, it consists of two objectives : for avoiding overload : possible ⇒ ((dev_i.measuredLoad > dev_i.$thLoad_{max}$) ⇒ (dev_i.estimatedLoad < **pre** dev_i.estimatedLoad)) for avoiding under-load : possible ⇒ ((dev_i.measuredLoad < dev_i.$thLoad_{min}$) ⇒ (dev_i.estimatedLoad > **pre** dev_i.estimatedLoad **or** dev_i.on = false)).

The deployment objective **d_obj$_{T4}$** is logico-numerical, as dev_i.measuredLoad is a numerical input and **pre** dev_i.estimatedLoad a numerical state.

3) Coordination objectives: from Section III-C.2 are

- **c_obj$_{T1}$**: QoStoProvide ≠ None ⇒ T_i.V_j.active, where T_i.V_i is a task version offering the requested functionality with the specified QoS;
- **c_obj$_{T2}$**: for each functionality f_i with a set of applicative objectives : a_obj_{ij}, **not** f_i.tasks.active ⇒ **not** (a_obj_{i1} **and** a_obj_{i2} **and** ... a_obj_{ij});
- **c_obj$_{T3}$** : T_i T_i.V_j.active ⇒ (T_i.V_j.objectsConfigurations.started **and** T_i.V_j.rules.activated **and** T_i.V_j.rules.objects ≠ None) for any of task T_i and its version V_j.

Once formalised, the objectives were defined in the *contract* part of a main H/BZR node. This node instantiated

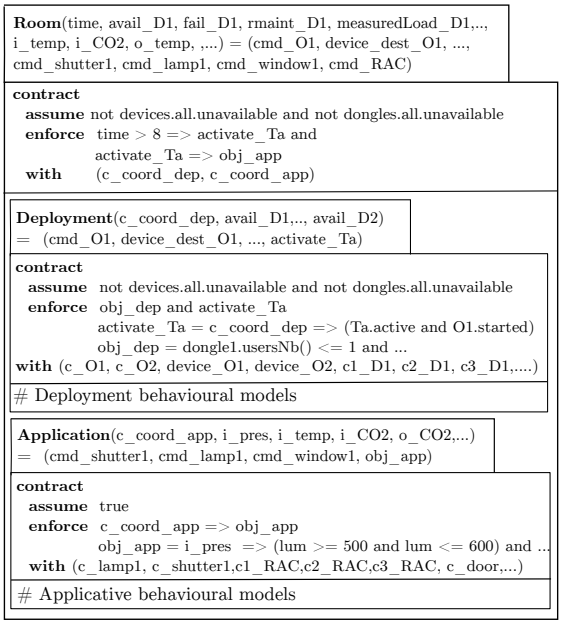


Fig. 8: Hierarchical node

and composed, in parallel, all the behavioural models and was compiled to generate one controller, for the whole system that is a building. To enable the generation of several controllers, that can be distributed, and reduce the synthesis cost, the control was done modularly and hierarchically.

C. Modularity and hierarchy

The system is first divided in two sub-systems: one for the applicative aspect and one for the deployment. Then, as shown in Fig. 8, a node with a *contract* is designed for each sub-system and a third node is built for their coordination:

- Applicative node: it composes the behavioural models of the applicative aspect and achieves the related objectives. This node has a set of inputs that are allows for its control by a node with a higher level of hierarchy. Such an input makes the applicative node achieve (or not) the objectives of the functionality related to it;
- Deployment node: it composes the behavioural models of the deployment aspect and achieve its objectives. This node has inputs that are allows for its control. Such an input makes the node activates a task and deploying its objects and rules, to provide a functionality, or stop it;
- Room node: it composes the applicative node, the deployment node and the behavioural models that are related to their coordination. It defines a *contract* in order to enforce the coordination objectives defined in Section III-C.2 with controllable variables that are inputs of the applicative and the deployment nodes.

To allow for different levels of hierarchy and considerably reduce the synthesis cost, each aspect of the system can be decomposed in sub-systems and controlled modularly. For instance, for the applicative aspect, the set of actuators are first divided into several sub-sets, depending one the target objectives. Then, the sub-sets are controlled as done in [16].

D. Dynamic reconfiguration of a controller

The controller of the system can be reconfigured, to deal with changing objectives. Indeed, a building, generally, have a set of configurations in which different objectives. For instance, the objectives to achieve during working time (e.g., comfort) are different from those to achieve during holidays (e.g., energy savings). To enable the reconfiguration, we propose a concrete solution which consists in first designing separately several controllers that achieve different objectives, and then, designing a reconfiguration controller that switches between the controllers. The switch consists in deactivating the current controller and activating the appropriate one upon the occurrence of specific events.

Activating a target controller requires ensuring that current state of the building is valid for this controller, for it to work. This is not an easy task because a state is valid for a controller only if it belongs to the controller state space, it does not violate its objective and, does not lead, through one or several uncontrollable transitions, to a state that violates an objective. For this, in the simplified case of our application domain, we first define a reconfiguration state the considered building. Then, we design the controllers of the building in such a way that they have the same initial state that is the reconfiguration state. This allows for activating any of the controllers when the building is in its reconfiguration state. Finally, we design the reconfiguration controller using H/BRZ. This is done by defining an automaton with a state for each controller of the building and and transitions among them. Such a state invokes the H/BZR node the corresponding controller. When this state is reached, the associated controller is automatically activated, all its variables and equations are reinitialised. This means that the controller starts in its initial state which is the reconfiguration state.

As an example, let us consider a building with two sets of objectives (working time, holidays) and a state `night` in which it is not occupied, not cooled, not heated and not ventilated. This state is first taken as the building reconfiguration state and a controller is designed for each set of objectives. Both controllers have `night` as initial state. The automaton defined to design to enable the reconfiguration of the controllers is presented in Fig. 9. It has two states (`WorkingTime`, `Holidays`) and two transitions. The initial state is `WorkingTime` meaning that the working time controller is the one that is active. When `holidays_started` is true and the `night` state is reached, the automaton goes to `Holidays`. This deactivates the `workingTime` controller and activates the `Holidays` one from its initial state that is `night`. This automaton is compiled and the generated `step` function is invoked in a LINC rule to enable its execution.

V. Physical implementation

The demonstrator presented in Fig. 10 was built to validate our contribution. It consists of an office that has a shutter (emulated by a graphical interface), a lamp (connected to a plugwise circle that allows for sending commands to it), a presence sensor (emulated by an EnOcean switch), an

Room(holidays_started, workingTime_started, night) = target_ctrl

assume not (holidays_started and workingTime_started)
enforce (holidays_started and night) => active_holidays_ctrl
(workingTime_started & night) => active_workingTime_ctrl
with (c1,c2)

active_holidays = Holidays_ctrl(c2);

active_workingTime = WorkingTime_ctrl(c1);

target_ctrl = if active_holidays then Holidays else if active_workingTime then WorkingTime else None;

Fig. 9: Reconfiguration controller design

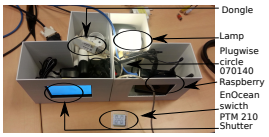


Fig. 10: Demonstrator

outdoor luminosity sensor, an agenda for meeting information (both are emulated by storing their values in LINC), a raspberry connected to two dongles for the communication with the circle and the switch (as an execution platform).

The objectives to be achieved in this office are (i) keep the luminosity between 500 and 600 lux when a presence is detected, (ii) keep the shutter closed during a confidential meeting and (iii) prefer natural lighting for energy savings.

a) Controller synthesis: Two automata, for the shutter and the lamp, and *contract*, define the luminosity and the confidentiality objectives. The energy savings objective is expressed by declaring the controllable variable of the shutter after the on of the lamp. This allows for using the lamp only when the office must be lighted and the shutter cannot be used. The generated controller was executed by on the Raspberry Pi. It takes as input the presence sensor data and information related to meeting (the outdoor luminosity is set to 600 lux) and executes commands of the lamp and shutter.

b) Controller behaviour: Fig. 11 presents the controller behaviour. The variables `time`, `i_presence` and `confidMeeting` respectively correspond to the time, the presence sensor value and an information specifying if there is a confidential meeting. The variables `shutter_opened` and `lamp_on` correspond to the state of the shutter and the lamp. At 7 AM, a presence is not detected in the office, the outdoor luminosity is equal to 600 lux (it is fixed) and there is no confidential meeting. In this case, the shutter is closed and the lamp is off. At 8, a presence is detected (one office member arrives), the controller opens the shutter to achieve both the luminosity and the energy savings objectives. At 11, the presence is still detected and there is a confidential meeting. In this case, the controller switches on the lamp and closes the shutter, for the confidentiality objective. At 12, the presence is not detected, it switches off the lamp.

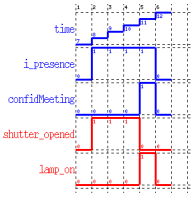


Fig. 11: Controller behaviour

VI. Conclusion

We present an application of Discrete Event Systems Control techniques to the domain of smart buildings in the context of IoT. Our contributions are: behavioural automata-based modelling, and objectives formalisation, for the coordination of the applicative and deployment managers of smart building systems; applications of Discrete Controller Synthesis to design monolithic or modular and hierarchical controllers; an implemented case study for a smart office.

Perspectives are in: exploiting the genericity of our models by designing a Domain Specific Language for developers to specify their systems, without needing expertise in formal models; exploring extensions in expressiveness in the control problems modelled and solved, for example advanced adaptive discrete control for reconfigurable controllers.

References

[1] N. Berthier, F. Alvares, H. Marchand, G. Delaval, and E. Rutten. Logico-numerical control for software components reconfiguration. In *IEEE Conf. Control Technology and Applications (CCTA)*, Aug 2017.

[2] N. Berthier and H. Marchand. Discrete Controller Synthesis for Infinite State Systems with ReaX. In *IEEE Int. Workshop on Discrete Event Systems*, Cachan, France, May 2014.

[3] G. Delaval, S. M. Gueye, and E. Rutten. Distributed execution of modular discrete controllers for data center management. In *Proc. 5th IFAC WS on Dependable Control of Discrete Systems, DCDS*, 2015.

[4] G. Delaval, É. Rutten, and H. Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, 2013.

[5] S. Guillet, B. Bouchard, and A. Bouzouane. Correct by construction security approach to design fault tolerant smart homes for disabled people. *Procedia Computer Science*, 21:257–264, 2013.

[6] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[7] D. Kolokotsa, A. Pouliezos, et al. Predictive control techniques for energy and indoor environmental quality management in buildings. *Building and Environment*, 44(9):1850–1863, 2009.

[8] M. Louvel and F. Pacull. Linc: A compact yet powerful coordination environment. In *Proc. Coordination Models and Languages*, 2014.

[9] M. Louvel, F. Pacull, et al. Development tools for rule-based coordination programming in linc. In *International Conference on Coordination Languages and Models*, pages 78–96. Springer, 2017.

[10] S. Mak Karé Gueye, É. Rutten, and J.-P. Diguet. Autonomic management of missions and reconfigurations in FPGA-based embedded system. In *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, Pasadena, USA, July 2017.

[11] A. K. Nabih, M. M. Gomaa, and oters. Modeling, simulation, and control of smart homes using petri nets. *International Journal of Smart Home*, 5(3):1–14, 2011.

[12] X. Niu and Z. Wang. A smart home context-aware model based on uml and colored petri net. *International Journal of Smart Home*, 10(1):101–114, 2016.

[13] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[14] A. N. Sylla, M. Louvel, and É. Rutten. Combining transactional and behavioural reliability in adaptive middleware. In *Proc. 15th International Workshop on Adaptive and Reflective Middleware*, 2016.

[15] A. N. Sylla, M. Louvel, and E. Rutten. Design framework for reliable and environment aware management of smart environment devices. *Journal of Internet Services and Applications*, 8(1):16, 2017.

[16] A. N. Sylla, M. Louvel, E. Rutten, and G. Delaval. Design framework for reliable multiple autonomic loops in smart environments. In *Int. Conf. Cloud and Autonomic Computing (ICCAC)*, 2017.

[17] M. Zhao, G. Privat, et al. Discrete control for the internet of things and smart environments. In *Presented as part of the 8th International Workshop on Feedback Computing*, 2013.